FAI Final Project Report

b09902133

June 2023

1 Methods

1.1 Heuristic Method

In this method, we design a function to evaluate the strength of the combination of our hole cards and the community card. The function evaluates it based on random simulation.

First, randomly sample some unseen cards and add them to the community cards such that the total number of community cards becomes five. Second, guess the hole cards held by each of our opponents by random sampling 2 unseen cards for each one. Then, evaluate the rank of the cards held by ourselves and each of our opponents using **HandEvaluator.eval_hand()**. If our score is higher than all of our opponents, we are considered to win this round of the simulation. The simulation will be run for 20 rounds. If we win more than 75% of the rounds, we will raise with the maximum amount. If we win more than 60% of the rounds, we will raise with the amount $m + \frac{(M-m)}{4}$, where M is the maximum possible amount and m is the minimum possible amount. If we win more than 50% of the rounds, we will call. Otherwise, we will fold.

1.2 Deep Q-Network (DQN)

In this method, we utilize Deep Q-Network proposed in [Mni+13] to decide the best action given the current state. The state can be represented as a vector containing some features, such as the pot amount, the amount of our stack, the rank of our cards evaluated by the heuristic mentioned in **1.1** and the current street. Besides, we consider 7 discrete actions, which are fold, call and raise with amount $\{m + i \cdot \frac{M-m}{4}\}_{i=0}^{4}$. (M is the maximum possible amount and m is the minimum possible amount). Given state vector s and action a, the value function $Q(s, a)$ is a Multilayer Perceptron (MLP) with 4 layers. In the Q-netork, the number of output nodes in the last layer is 7, where the $i-th$ node of the last layer represents the Q-value according to the $i-th$ action.

Before the training starts, we initialize a Q-function Q and a target Q-function $\hat{Q} = Q$. We play 1000 games with baseline5 and train the Q-network during the games. At training step t, we takes action according to ϵ-greedy policy. Namely, randomly choose an action with the probability ϵ and choose the action based on $argmax_a Q(s, a)$ with the probability $1 - \epsilon$. Then, we store the transition record (s_t, a_t, r_t, s_{t+1}) to the replay buffer, where s_t is the state at step t, a_t is the action taken at step t, r_t is the amount of our stack increment at step t and s_{t+1} is the state at step $t + 1$. Then, randomly sample a mini-batch of transition records (s, a, r, s') to train the Q-network. The optimization goal is the temporal difference (TD) error which can be represented as the following formula:

$$\mathcal{L} = \mathbb{E}(r + \gamma max_{a'} Q(s', a') - \hat{Q}(s, a))^2$$

The gradients will only propagate to Q and not propagate to $\hat{Q}$. At every C steps, copy the parameters of Q to $\hat{Q}$.

At inference stage (the agent used in competition), the policy $\pi(s) = argmax_a Q(s, a)$. We always choose the action based on this policy.

2 Configurations

2.1 Heuristic Method

The hyper-parameters and configurations have already been described in section **1.1**.

2.2 Deep Q-Network

We use Adam optimizer to optimize the Q-network. The learning rate is set to 0.001, and the batch size is set to 16. We set $\epsilon = 0.3$, $C = 8$ and $\gamma = 0.99$. Before the training starts, we let our agent randomly takes action for 20 steps without updating the parameters of the Q-network (warm-up steps).

The Q-network consists of 4 fully-connected layers. The number of input features is 4, and the number of output nodes at each layer is 256, 128, 64 and 7 respectively.

3 Comparison

We compare the performance of the two methods based on some criterion. First, We let our two agents play 100 games with each of the 5 baseline agents and the random agent respectively. Then, we compare the win rate of the two agents.

	random	baseline 1	baseline 2	baseline 3	baseline 4	baseline 5
Heuristic Agent	0.41	**0.67**	0.45	0.62	0.23	0.22
DQN Agent	**1.00**	0.53	**0.57**	**0.75**	**0.45**	**0.50**

In addition, we let the two agents play 100 games with each other. The DQN agent won **74** times in total, while the heuristic agent won **26** times in total.

4 Discussion and Conclusion

We found that the DQN agent outperformed the heuristic agent against random agent, baseline2 agent, baseline3 agent, baseline4 agent and baseline5 agent, while the heuristic agent outperformed the DQN agent against baseline1 agent. Moreover, the DQN agent won significantly more times than the heuristic agent did when the two agents competed with each other.

The heuristic agent usually takes conservative actions. It only raises the amount of bet when its confidence of having better combination is high. However, the DQN agent sometimes takes adventurous actions. The randomness of action selection in the training process of DQN lets the agent learn to evaluate the consequence of risky actions based on the current state information. The complexity in the decision process of the DQN agent makes it preform better in general.

However, there are also some potential drawbacks in the DQN agent. For example, Texas Hold'em is a game with high randomness, which makes it difficult to learn a general and robust DQN. Although we take completely the same action, the transition of states may be different. This makes the training of DQN unstable and hard to converge. Besides, the current model does not consider the history of actions taken by our opponents. However, action history might be a useful information since we can guess the hole cards held by our opponents by observing their previous actions. Moreover, DQN can not handle continuous actions, which makes the actions taken by the DQN agent less accurate.

5 Code Reference

1. https://github.com/ishikota/PyPokerEngine
2. https://github.com/hungtuchen/pytorch-dqn

References

[Mni+13] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: `1312.5602 [cs.LG]`.